IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received August 23, 2019, accepted September 24, 2019, date of publication September 27, 2019, date of current version October 11, 2019.

Digital Object Identifier 10.1109/ACCESS.2019.2944238

A Method to Construct Task Scheduling Algorithms for Heterogeneous Multi-Core Systems

SUNG IL KIM AND JONG-KOOK KIM, (Senior Member, IEEE)
School of Electrical Engineering, Korea University, Seoul 02473, South Korea

Corresponding author: Jong-Kook Kim (jongkook@korea.ac.kr)

This work was supported in part by the National Research Foundation of Korea (NRF) through the Basic Science Research Program funded by the Ministry of Education under Grant 2014R1A1A2059527, and in part by the Information Technology Research Center (ITRC), Ministry of Science and ICT (MSIT), South Korea, through a Support Program under Grant IITP-2018-0-01433, supervised by the Institute for Information and Communications Technology Promotion (IITP).

ABSTRACT The use of heterogeneous multicore processors (HMP) is spreading rapidly from data centers to large-scale deployment in smartphones because they give greater flexibility to adapt to power constraints and performance needs. In this study, we show that an intelligent task scheduler is critical for improving the performance and energy efficiency in an HMP environment. We assume that the tasks are independent in the environment, with hard real-time constraints and multicore systems, where the processors can be manipulated to change the clock cycle speed and power levels. Tasks are assumed to arrive aperiodically where the tasks are applications from the SPEC CPU 2006 benchmark suite. In the evaluation, we used a real system comprising of two multicore processors, which supported on-the-fly dynamic voltage/frequency scaling. We extracted several important components from previously proposed algorithms and combined them to construct algorithms with better performance. Our results showed that some of the best combinations reduced the energy consumption and achieved a better completion rate in the environment. In addition, a method is proposed for calculating the upper-bound of the task completion rate and energy consumption so that there is a guide as to how near the results are to the optimal performance.

INDEX TERMS Task scheduling, energy-aware scheduling, heterogeneous multi-core, real-time scheduling, dynamic voltage/frequency scaling.

I. INTRODUCTION

Constructing a processor using multiple simple cores rather than a single complex core is now a mainstream practice because processors with multiple cores have a higher throughput and better power efficiency. These multicore processors are used in data centers on a large scale to reduce power consumption because a large proportion of the cost incurred while maintaining a data center is attributable to utility costs. In addition, mobile phone vendors now manufacture smartphones with multicore processors because their energy efficiency can reduce power usage to prolong the device's use before recharging.

A better solution may be a heterogeneous multicore processor (HMP), which allows opportunities for saving power and using computational resources effectively. The heterogeneity can be found in all parts of a system and difference in the processor includes the core architecture, cache capacity, clock cycle frequency, and the amount of memory for each core. An HMP may enhance the throughput and power efficiency when applied to targeted hardware design [2]. For an HMP system, the assignment of tasks (or jobs) to appropriate cores is a critical issue for maintaining high throughput and energy efficiency. Tasks may be time-critical in terms of their response time, user experience, or completion time.

There are two different types of scheduling: static and dynamic. Static scheduling is performed when the applications/tasks are mapped in an off-line planning phase (Their arrivals and information about the applications/tasks are known before the scheduling). As the time constraint to determine a ''good'' scheduling for a off-line planning phase or analyzing phase is not strict, schemes that may run for a long time to produce a good scheduling can be used.

The associate editor coordinating the review of this manuscript and approving it for publication was Sun-Yuan Hsieh.

Static scheduling algorithms are usually based on meta-heuristic algorithms such as generic algorithm (GA), ant colony optimization (ACO), and bee colony algorithm. (e.g., [30], [31] [32], [34]) Job shop scheduling (e.g., [18]–[21]) is an active area of task scheduling research, where the schedule for a set of production jobs is planned before the production and this scheduling can be done with static scheduling schemes. Dynamic scheduling is performed when the applications are mapped in an on-line fashion, (e.g., when tasks' arrival are unpredictable) and are mapped as they arrive (the workload is not known a priori). In both cases, the scheduling problem has been shown, in general, to be NP-complete [34]–[36]. Thus, the development of heuristic techniques to find near-optimal solutions for the scheduling problem is important for the efficient use of computing resource.

In this study, we designed and analyzed dynamic scheduling algorithms for an example hard real-time HMP system. The HMP system is comprised of two quad-core processors, where the clock cycle speed and energy usage could be changed using dynamic voltage/frequency scaling (DVFS) [4]. DVFS is employed widely to reduce the power consumption of processors by lowering the supply voltage and/or the clock cycle speed. However, there is a trade-off between the speed of execution and the energy being saved, so the task scheduler must intelligently allocate resources to tasks while using the appropriate voltage or speed level. The HMP system considered in this study provides per processor on-the-fly (during runtime) DVFS such that frequency scaling and task scheduling can be performed dynamically. Using a threshold, one processor operates at the lower frequency levels while the other processor operates at the higher frequency levels. Thus, the heterogeneity of the system is made by clock frequency. The task requests arrive to the system aperiodically and the application that needs to be executed (i.e., the task) is selected randomly from the 29 applications in the SPEC CPU 2006 benchmark suite [12]. In this study, we assumed that the task requests are independent, non-preemptive, and they have hard real-time constraints (deadline or response time). A task is considered to be completed when the task is completed by its deadline, whereas task is failed when the task is completed past its deadline, and the task can be dropped when execution is not attempted and the task is no longer considered for execution. This research assumes an oversubscribed system where tasks may be dropped because tasks may not meet the deadline. Therefore, for each task, it is determined whether it can be completed before its deadline before its execution. If the task cannot meet its deadline, the task will be dropped. In this environment, using a naive method may incur a lot of tasks to be dropped. Thus, using an intelligent method for task scheduling is needed to minimize task drops. In many systems, there is no gain for the completion of tasks that does not satisfy their deadline.

There are many existing works in the literature of heterogeneous platforms. However, the environment we assumed is different from previous ones and we wanted to test our method for constructing scheduling schemes in an actual environment that is setup using commodity server. The main contributions of this study are that we analyzed the algorithms used in our previous work [9] as well as other basic scheduling methods to determine the components that are important for task scheduling, and we used these components to construct various scheduling algorithms. This method for determining the important components of task scheduling in a particular environment and the best combination of components for building a ''good'' task scheduling algorithm has not been described previously to the best of our knowledge. These various combinations of components are tested and we finally determined the best algorithm for the proposed environment.

The remainder of this paper is organized as follows. Related research is discussed in the next section. In Section III, the system models are presented. Section 4 describes the detailed task scheduling algorithms. The evaluation results and their analysis are presented in Section V. In Section VI, we give our conclusions.

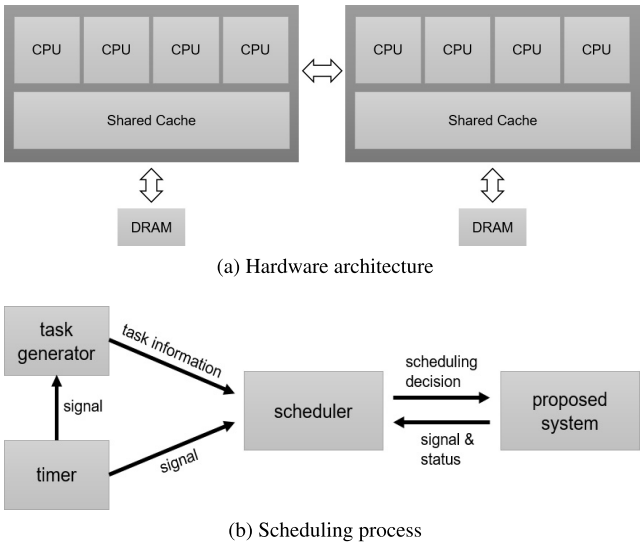
II. RELATED WORK

Many studies have addressed task scheduling for HMP under real-time constraints. Enhancing the performance of non-real-time tasks in the presence of a real-time workload was considered by Calandrino et al. [15] to allow the scheduling of periodic soft real-time tasks. For periodic task, Moulik *et al.* [23] proposed a heterogeneous energy-aware real-time scheduler which has three level of hierarchical resource allocation. The scheduler first computes a set of fragments of the execution and schedules each task in order to allow their appropriate execution share, and configure the operating frequencies in each core to minimize energy consumption. They devised an improved algorithm [24] taking system-wide energy consumption into consideration with better optimal frequency selection method. Chwa *et al.* [26] designed fully-migrative approach to two-type heterogeneous multicore scheduling called *Hetero-Fair*. Moulik et al. extended *Hetero-Fair* algorithm to be applicable to generic heterogeneous platforms and enables it with a low-overhead [27], [28]

Tang *et al.* [16] introduced a task scheduling algorithm for a combination of hard periodic and soft aperiodic real-time tasks. The periodic tasks are scheduled by an offline scheme and the aperiodic tasks are scheduled dynamically using the remaining slack time information for each resource. Lin et al. proposed a processor mapping algorithm based on the integer linear programming (ILP) model for real-time streaming systems [17]. To facilitate global optimization of the throughput, latency, and processor cost, a global ILP model was proposed and appropriate solutions were found using a version of genetic algorithm. By contrast, our method considers the real-time characteristics of the system as well as its energy characteristics and the randomness of task requests or arrivals. We use DVFS to reduce energy usage and tested scheduling

algorithms on a real system using benchmark applications as tasks. Task scheduling algorithms that consider energy have been proposed for many heterogeneous multicore systems. Energy minimizing processor allocation for periodic real-time tasks was proposed by Chen *et al.* [13], where the solution to the ILP model is derived by a greedy approach to minimize the energy consumption. Wenjing and Lisheng [14] designed a task scheduling algorithm that considers both the execution time for tasks and energy consumption in an ILP model. A task is modeled as graphs and it is assigned the highest priority level when it has the shortest execution time and the lowest energy consumption. Scheduling is performed in order of priority. Yu *et al.* [10] designed an ILP-based static resource allocation algorithm, where the voltage levels of processing elements are considered as a parameter in the ILP problem formulation for energy reduction. Zhang *et al.* [1] implemented the shuffled frog leaping algorithm (SFLA) for real-time periodic task scheduling, where the objective is minimizing the energy consumption while satisfying the task deadline. Hing *et al.* [11] describes an energy-efficient scheduling algorithm for a dynamic voltage scaling (DVS) system and a non-DVS processing element, where the objectives are minimizing energy consumption and maximizing energy savings. Power consumption is modeled by two different cases, where the non-DVS processing unit is either workload-dependent or not workload-dependent. The proposed algorithms are approximation algorithms based on ILP. Also, Baruah *et al.* [22] proposed ILP formulation based partitioning approach for constrained-deadline tasks on heterogeneous processors. Awan *et al.* [29] addressed task allocation on DVFS enabled heterogeneous multicore platforms which minimize overall energy consumption of the system taking leakage energy and dynamic energy into consideration. They provide a task-to-core mapping algorithm and determine sleep states and operating frequency-level of each core for a given set of independent sporadic tasks. Lin *et al.* [3] proposed an energy-efficient task scheduling method that operates under two execution modes: batch mode and online mode. In the batch mode, a greedy based method is used to minimize the total cost, where the cost comprises the time and energy consumption, processing rate (or clock frequency level), the cost of energy, and the amount paid for waiting by users. In the online mode, a new task is generated continuously and introduced to the system. The tasks belong to two types in the online mode: interactive and non-interactive tasks. Interactive tasks demand a short response time, whereas non-interactive tasks do not focus on the response time (or makespan in this case). Therefore, an algorithm for the online mode is used to complete interactive tasks with a short response time and to minimize the non-interactive task completion time.

In contratst, we use DVFS to reduce the energy usage and tested scheduling algorithms an intelligent scheduling method to complete as many tasks as possible. We assume that all tasks arrive randomly, with hard deadlines, and they are non-periodic.



(a) Hardware architecture

(b) Scheduling process

FIGURE 1. Architecture of the environment and the scheduling process.

III. SYSTEM MODEL

A. SYSTEM ARCHITECTURE

The HMP system considered in this study uses two quad-core processors with per processor DVFS enabled. The frequency level of the processors can be determined by the user, or dynamically by the system. The processors are connected by an off-chip interconnection and there is main memory per processor. In terms of memory access, this environment is a non-uniform memory access (NUMA) architecture. Fig. 1a illustrates the architecture of the hardware used in this study.

Our system assumes a fast/slow processor architecture, where both processors are identical except for their clock frequency level. The fast processor has high frequency range and the slow processor has low frequency range.

Fig. 1b shows the overall scheduling process for the proposed heterogeneous multicore system. The timer sends a signal to the task generator and the scheduler, which follows a Poisson distribution with a mean value of m. At each signal, the task generator generates tasks. The scheduler receives this signal as a task arrival signal and it is ready to receive information for newly arrived tasks from the task generator. When the task is received by the scheduler, the appropriate frequency level and core assignment is determined for the task using information related to the runtime environment, such as the core and queue status. After the assignment is complete, the assigned tasks are inserted into the private queue on each core. The tasks in the queue can be moved freely to other queue position, including their own queue or other queues. After the task has been completed, a task execution completion signal is sent to the scheduler, and thus another scheduling event occurs after task completion. These signals used are implemented by system calls in linux.

B. TASK

The task model is the same as the one used in our previous study [9]. The task requests are selected randomly from

TABLE 1. Notation.

Notation	Description
$task_i$	task i
$task_{i,j}$	task i on core j
d_i	deadline of task i
$ET_{i,j}$	expected execution time for task i on core j
$Ready_j$	remaining time for executing the task on core j
$CT_{i,j}$	expected completion time for task i on core j

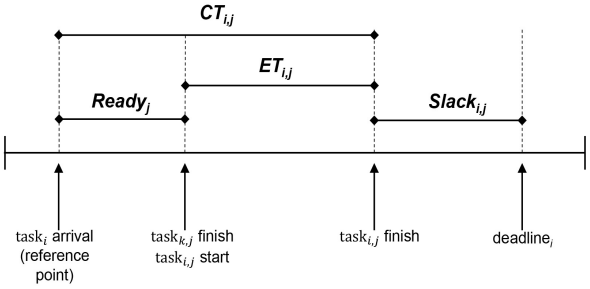


FIGURE 2. Terms for real-time task scheduling.

29 applications in the SPEC CPU 2006 benchmark suite. The SPEC CPU benchmark is the most popular benchmark suite for measuring the performance of various CPU architectures. The suite comprises a set of 29 computationally intensive applications for measuring the performance in integer and floating point calculations. The mean time interval between the tasks is set to 6 seconds using a Poisson distribution for this environment. The mean time interval is empirically determined to maintain a system that is oversubscribed while the Linux scheduler could complete around 27% of the total number of tasks. The mean inter-task arrival time is not known in advance. Therefore, the arrival time for the task requests is not known to the system. All tasks have a hard deadline, which is two times longer than the estimated execution time (EET, which is the average actual execution time) using the slowest frequency level to give tasks the opportunity of being completed successfully before their deadlines. The EET is an empirically predetermined average value when a single application is executed on the system 10 times. The group of tasks is assumed to be known (29 applications from the benchmark suite), therefore the tasks are executed using all frequency levels and the information is used by the system. However, this information does not consider the concurrent execution of different or same multiple task combinations.

Fig. 2 shows the timeline for the execution of $task_k$ on $core_j$ when $task_i$ first arrives in the system. (The notations and definitions are shown in Table 1. We assume that $core_j$ is the only available core for task assignment. When $task_i$ arrives, the expected execution time for $task_i$ at each frequency level is announced. If no task exists in the queue except for $task_i$, $task_i$ must wait for the completion of $task_j$. This waiting time is defined as the ready to execute time ($Ready_j$). The expected completion time of $task_i$ on $core_j$ ($CT_{i,j}$) is the sum of $Ready_j$ and $ET_{i,j}$. $ET_{i,j}$ is known when $task_i$ arrives, and thus $CT_{i,j}$ is also known at the arrival time for $task_i$.

In the evaluations, a task is considered to be completed when the task has finished execution by its deadline, whereas a task is failed when it is completed past its deadline. Tasks that were not executed at all are considered to be dropped. In this research, the task completion percentage, total energy consumption and energy consumption per task completion are used as performance metrics.

C. ENERGY MODEL

The energy model focuses on the energy consumption by the processor. Using the power model described by Winter *et al.* [6], the equation for dynamic power consumption by the processor can be denoted as

$$P_{dynamic} = CV^2f \tag{1}$$

where C, V, and f denote the capacitance of the processor, the supply voltage, and the clock frequency of the processor, respectively. To simplify the problem, it is assumed that a linear increase in the clock frequency leads to a linear increase in the supply voltage, and vice versa. If this notion is applied to Equation 1, then it results in the following.

$$P_{dynamic} \propto V^2 \tag{2}$$

The energy consumption during task execution is equal to the execution time multiplied by the power consumption.

$$Energy_{execution} = P_{dynamic} \times execution\ time \tag{3}$$

When the processor is in the idle state, the processor consumes less power than that in the operating state. According to Kim *et al.* [9], when it is in the idle state, an Intel Xeon E5620 processor consumes about half the operating power consumed during task execution. Thus, the energy consumption by an idle processor is as follows.

$$Energy_{idle} = P_{idle} \times idle\ time \tag{4}$$

The total energy consumption is determined as the sum of the task execution energy and the idle processor energy consumption.

IV. SCHEDULING ALGORITHMS

A. OVERVIEW

In this section, we introduce various components of the task scheduling algorithms investigated and extracted from a previous study [9]. A scheduling algorithm can be designed by choosing different options from each of the scheduling components. The private queues described will automatically drop tasks that have exceeded their deadlines while waiting in the queue. In addition, the task at the front of the queue is checked before its execution to determine whether the task can be executed by the deadline or not. If it is determined that the task cannot be completed before the deadline, then the task is dropped immediately before its execution.

B. SCHEDULING COMPONENTS

1) TASK CLASSIFICATION

The aim of a heterogeneous architecture is exploiting heterogeneity via the intelligent assignment of tasks and processing elements; therefore, task classification may be important. The environment considered in this study assumes a fast/slow processor architecture, and the tasks are classified as short and long tasks. The task classification component employ three options: average, average of the tasks in the queue, and accumulated average. The threshold that divides short and long tasks can be predetermined before the simulation or determined during runtime.

Average: A simple solution is to determine the average task execution time of all tasks and use it as the threshold. If the execution time is known for all tasks, then the average execution time that determines short/long tasks can be predetermined.

Average of tasks in the queue: If the execution times of all tasks are not known in advance, the classification method must determine the threshold dynamically from information given through the runtime environment. The average of the tasks in the queue is determined based on the average execution time of all the tasks currently in all of the queues.

Accumulated average: The accumulated average option uses the execution time of all tasks that have arrived (i.e., execution time of tasks that have completed are also used).

2) PROCESSOR ASSIGNMENT

The tasks are divided into two categories, therefore the processor assignment process also has two options. After the task has been assigned to a fast or slow processor, core assignment is performed using the minimum completion time method. The task completion time is calculated as the sum of the ready-to-execute time and the expected execution time for the task. ($CT_{i,j} = Ready_{i,j} + ET_{i,j}$) The minimum completion time method assigns a task to the core that minimizes its completion time.

SFLS: "Short task to the fast processor, long task to the slow processor." The rationale behind assigning a short task to the fast processor is based on performance, whereas a long task is assigned to the slow processor for low energy consumption. The disadvantage of SFLS is that a slow processor may suffer from bottlenecks due to an excessive number of long tasks.

SSLF: "Short task to the slow processor, long task to the fast processor." SSLF has an advantage for completing long tasks because executing long tasks on a fast processor significantly reduces the execution time for long tasks.

3) QUEUE ORDERING

The execution order is also important. We evaluated two widely used ordering methods in this study.

EDF: Earliest deadline first. The task with the nearest deadline is sent to the front of the queue.

TABLE 2. Benchmark execution time (in seconds) at highest frequency.

Application	Execution time	Application	Execution time
perlbench	29.25	povray	9.64
bzip2	57.37	calculix	1.70
gcc	1.11	hmmer	59.05
bwaves	220.39	sjeng	148.18
gamess	158.54	GemsFDTD	60.95
mcf	19.66	libquantum	2.23
milc	18.85	h264ref	106.27
zeusmp	45.50	tonto	284.35
gromacs	164.65	lbm	42.65
cactusADM	34.03	omnetpp	63.99
leslie3d	151.92	astar	126.67
namd	17.16	wrf	196.43
gobmk	118.99	sphinx3	10.70
dealII	35.16	xalancbmk	71.32
soplex	4.05		

SRF: Shortest remaining time first. The tasks in the queue are sorted by execution time in increasing order, so the shortest task is executed first.

4) TASK MIGRATION

In this paper, task migration means the movement of a currently executing task to another processor. To maintain the load balance in the system, we only permit migration from a high load processor to a low load processor which has an idle core(s). Thus task migration does not violate non-preemptive assumption of this research. The processor load denotes the total execution time for the tasks on the processor, which includes the remaining execution time for the currently executing tasks and the overall task execution time for the processor queue. In our previous study, task migration was not considered, and thus this is a new component for this paper.

SRFP: Shortest remaining time task to a fast processor. Task migration to a fast processor is allowed. If an idle core exists on a fast processor, the task on a slow processor with the shortest remaining time is migrated to the idle core on the fast processor.

LRFP: Longest remaining time task to a fast processor. This migration option is the same as SRFP, but the longest task is selected for migration for a task running on a slow processor.

SRSP: Shortest remaining time task to a slow processor. SRSP only allows task migration to a slow processor. The task with the shortest remaining time is selected for migration.

LRSP: Longest remaining time task to a slow processor. The task with the longest remaining time on a fast processor migrates to a slow processor.

5) DVFS

Seven frequency levels are available in this environment for DVFS. In this paper, we use Westmere CPU family for evaluation. According to [25], the maximum frequency transition latency of Westmere CPU is 65.48 μs. This latency is negligible compared to the execution time of applications. Table 2 shows execution time of benchmark at highest frequency

level. We evaluated execution time of applications at each level and exploited them to estimate execution time of the application at designated frequency level.

Step: The step option moves the clock frequency up one level to achieve a better performance. The clock frequency can be downgraded by one level to reduce the energy consumption. CTPR uses this option for frequency scaling.

Leap: Leap controls the clock frequency according to two levels: the highest and lowest frequency levels which are different for different processors. The aim of the leap option is to try to obtain greater energy savings if the task can be completed by its deadline even when the lowest frequency level is applied.

6) TASK STEALING

If the algorithm assigns short tasks to fast processor, one or more of its cores may become idle after the rapid execution of a short task. When a fast core is idle and its queue is empty, the fast core steals a task with the minimum execution time among all the queues in the system. The task stealing option employed in this study follows this method but it is not restricted to fast core processors. Thus, if an idle core exists and its queue is empty, task stealing occurs. It is expected that task stealing will increase core utilization by resolving the task starvation problem. In addition, the smallest task is selected for execution to prevent a long waiting time for a task that arrive during the execution of the selected task.

C. UPPER BOUND AND ENERGY BOUND

The upper bound is calculated for the task completion performance. The dynamic upper bound [8] assumes the task arrival information is not known a priori. We modified the dynamic version to a static version assuming the task arrival information is known in advance. Same as dynamic upper bound, the concept of the total aggregate computational time (TACT) is used for the static upper bound. The method employs the time interval between the arrival of tasks, TACT is determined by multiplying the inter-arrival times of the task, and the number of processing cores, where the task is executed based on the TACT. In other words, if the interarrival time of consecutive tasks ($task_i$ and $task_{i+1}$) is k seconds and the number of processing cores is n, TACT is $k \times n$. That is, the system has TACT amount of time to execute the tasks in the system. The task execution time may be different among cores, but the lowest execution time (execution time using the lowest clock frequency level) is selected for calculation.

First, calculate TACT from each task arrival, which we denote as TACT slots. Select the task with the lowest execution time. Execute the selected task by deducting TACT, and if there is still time remaining, the following TACT slots may be used. If the task cannot be completed until the last TACT slot, the task is assumed to be dropped and not executed at all. Thus, the remaining TACT will be returned. In addition, this method is applied to calculate the lower bound for energy consumption. As executing no tasks is the absolute lower bound, the energy bound can be calculated only when the task completion rate is fixed. That is, when there is a target task completion rate is determined, the energy bound for that rate is calculated.

Algorithm 1 Algorithm for Energy Bound

Required: task arrival time, task execution time on slow cores, execution energy of cores, idle energy of cores, target task completion ratio

1: Determine the total aggregate computation time (TACT) for every slot by multiplying each time interval between task arrivals and the number of processing cores, where the i^{th} slot is denoted by TACT(i).
2: Using the global information for tasks, make a task list by sorting the tasks by execution time (ascending order), where the k^{th} task on the task list is denoted by task(j, k) (j denotes for j^{th} arriving task).
3: Initialize rETC(j, k) (remaining execution time to complete task(j, k)) for all tasks based on the maximum execution time over all cores.
4: Select the task(j, k) with the shortest execution time from the task list, where i = j.
5: **if** TACT(i) $\leq$ rETC(j,k) **then**
6: store i and value of rETC(j,k) (prepare task drop case)
7: rETC(i, j) = rETC(i, j) - TACT(i)
8: TACT(i) = 0
9: **if** next TACT slot exists **then**
10: move to next TACT slot (i = i + 1)
11: repeat step 5
12: **else**
13: drop task(j, k)
14: during the evaluation of task(j, k), the value of rETC(j, k) and TACT(i, i $\geq$ j) are reduced. Recover rETC(j, k) and TACT(i, i $\geq$ j) with stored values.
15: **end if**
16: **else**
17: TACT(i) = TACT(i) - rETC(i, j)
18: rETC(i, j) = 0
19: add execution energy of task(j, k) to total execution energy
20: remove task(j, k) from task list
21: get current task completion ratio
22: **end if**
23: Repeat steps 4−5 until no task exists on the task list OR current task completion ratio reaches the target task completion ratio.
24: Set j as the largest value of the arrival order based on completed tasks.
25: Calculate idle energy from TACT(i, i = 1) to i = j.
26: Evaluate total energy consumption. (= idle energy + execution energy)

V. EVALUATION

A. EVALUATION METHOD

The hardware shown in Table 3 was used for the evaluation. The Intel Xeon E5620 supports simultaneous multithreading

TABLE 3. Hardware configuration.

CPU - Intel Xeon E5620	
Core count	4 ×2
Clock speed	1.6Ghz to 2.4Ghz (seven frequency levels)
Cache	12MB Shared cache ×2
Mainboard - Supermicro X8DTi	
Chipset	Intel 5520 (Tylersburg)
RAM - Samsung DDR3 4G PC3-10600 ECC/REG	
Clock speed	1333Mhz

(hyperthreading [7]) but we disabled multithreading to ensure that one task executes on a core. The processor also supports Enhanced Intel Speedstep Technology [5], so the processor can have seven distinct frequency levels and the clock frequency is controlled on a per processor basis. The environment assumes that the fast processor uses the upper 3 frequency levels, whereas the slow processor utilizes the lower 4 levels. The 29 applications in the SPEC CPU 2006 benchmark suite are used as tasks and 64-bit Linux was used as the system OS. Without core assignment, the task floats among cores while running in linux. Therefore, we used the taskset command in linux to make a task execute on the designated core. The system and algorithms were implemented using C/C++ and bash shell script. In order to obtain the estimated execution time, experiments were conducted to measure the execution times for all the applications at each frequency level using the same system. The execution time was measured 10 times and the average value was recorded for use by the system. The execution time of the applications varied from 1.62 seconds to 292 seconds. The task arrival time was determined using a Poisson distribution with a mean interval time of 6 seconds. Each of the scheduling methods were run five times and the results were averaged. The evaluation time of the systems was 2 hours and this generated around 1200 tasks in the system for one trial of the simulation. To calculate the energy consumption, a baseline power unit was defined with respect to the power consumption of the lowest frequency level (1.6 GHz), which was set as 1 unit per second. The power consumption by the other levels was determined assuming that the increase was linear. The energy consumption was calculated by multiplying the power consumption of a core by the execution time on that core. The scheduling algorithms employed in the evaluation comprised combinations of scheduling component, with a total of 540 combinations. Evaluating all 540 scheduling algorithms multiple times would have been too time consuming, therefore each combination of the scheduling components are tested once and excluded the options with very low performance. The names of the algorithms are constructed using the options shown in Table 4. Instead of providing a long list of the scheduling components and their options, we simply used the initial of each component and the index of the option. For example, using the average for task classification, SFLS for processor assignment, SRF for queue ordering,

TABLE 4. Algorithm naming.

Task classification(C)		Processor assignment(P)	
Option	Index	Option	Index
Average	1	SFLS	1
Average(queue)	2	SSLF	2
Acc. average	3		
Queue ordering(Q)		**DVFS(F)**	
Option	Index	Option	Index
EDF	1	Step	1
SRF	2	Leap	2
Task migration(M)		**Task stealing(S)**	
Option	Index	Option	Index
SRFP	1	Stealing	1
LRFP	2		
SRSP	3		
LRSP	4		

SRFP for task migration, leap for DVFS, and task stealing is simply named as C1P1Q2M1F2S1. When there was no option, we eliminated the initial of the scheduling component. For example, if the algorithm does not use task migration component, then there is no 'M' character in its name. This method of naming of the scheduling option is used throughout the paper and on the graphs as well.

B. EVALUATION

540 combinations are evaluated and analyzed, not all of the results are shown due to space limitations. We compared our best combinations with previous algorithms. Energy bound is also presented.

1) TASK CLASSIFICATION

Fig. 3 shows the performance of each option of task classification components based on the P1 scheduling combination. The average achieved the best performance and the performance of the accumulated average was slightly lower. The reason is that the task execution time was unstable in the queue, which changed the task classification threshold drastically. In our experiments, the execution time of the accumulated average converged in the latter part of each trial. Task-in-the-queue-based options exhibited significantly lower performance.

2) PROCESSOR ASSIGNMENT

The tasks were classified before being assigned to the processor. We evaluated the two processor assignment methods described in Section IV. Fig. 4 shows the task completion rate for the processor assignment methods (selected combinations). SFLS performed better in most cases, which indicates that the fast execution of short tasks (SFLS) is more profitable than shortening the execution time for long tasks (SSLF). The numbers of completed tasks are shown in Fig. 5. Cores 0-3 are slow cores, and cores 4-7 are fast cores. SSLF assigns short task to slow cores, which makes slow cores complete many

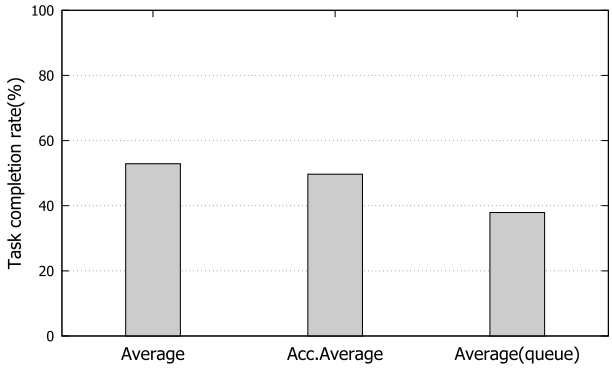


FIGURE 3. Performance of task classification method P1.

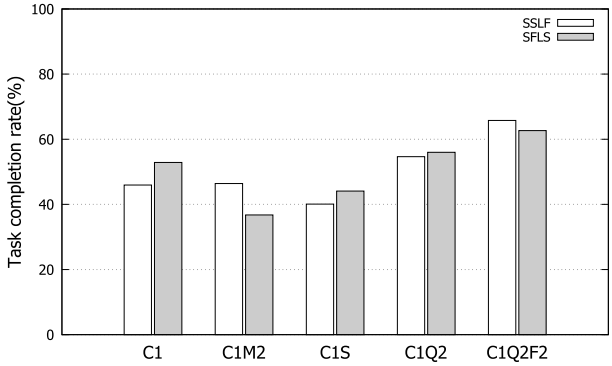


FIGURE 4. Processor assignment method.

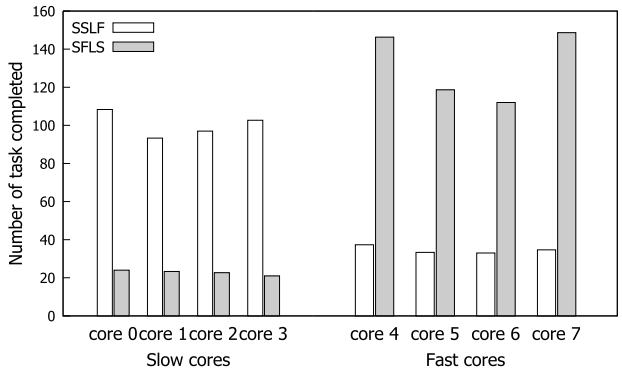


FIGURE 5. Number of task completed on cores.

short tasks. SFLS completed a higher number of short tasks (cores 4-7) than SSLF (cores 0-3). Due to the higher number of short tasks completed, SFLS performed better, although the number of long tasks completed was lower than that of using SSLF. In some cases, SSLF performed better than SFLS such as the C1M2 and C1Q2F2 combination.

3) QUEUE ORDERING

Three options were compared, including no option. EDF was better than no option, but SRF achieved the best performance, as shown in Fig. 6. SRF used less energy with a higher task completion rate. The deadline of a task is determined by its execution time (or correlated), so EDF and SRF are similar options. However, the difference can be explained by

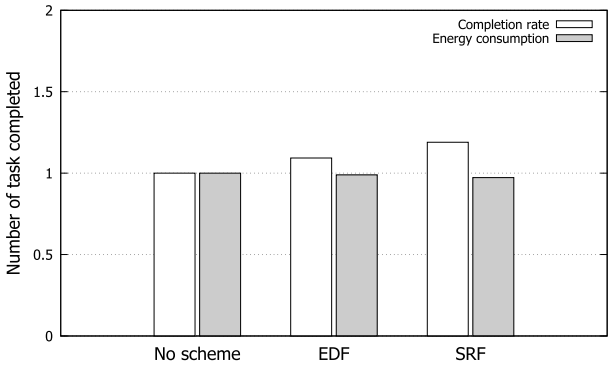


FIGURE 6. Performance of queue ordering on C1P2.

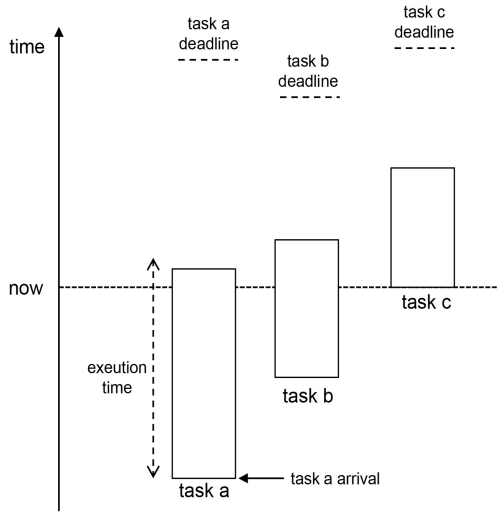


FIGURE 7. Example to show difference between EDF and SRF.

an example. Fig. 7 illustrates the sequential arrival of task *a*, task *b*, and task *c*. Let's assume that task *c* has just arrived and the core has become idle. EDF orders the tasks according to their deadline, so the order is task *b*, task *a*, and task *c*. In contrast, for SRF, the order of tasks is task *c*, task *b*, and task *a*. In this case, either task *b* or task *c* can be completed, whereas the other task will be dropped due to the deadline. EDF will execute task *b*, whereas SRF will select task *c* for execution. As shown by the results, SRF performed better because it selects a shorter task for execution. According to the example, the completion time of executing task *c* was earlier than that for executing task *b*.

4) TASK MIGRATION

The task migration methods differed in performance according to the processor assignment method. SFLS assigns a short task to a fast processor and a long task to a slow processor, and it performed better with SRSP or LRSP as shown in. Fig. 8. SSLF assigns a short task to a slow processor and a long task to a fast processor, and it performed better with SRFP or LRFP as shown in Fig. 8. SRSP and LRSP migrate short tasks or long tasks onto the fast processor or the slow processor

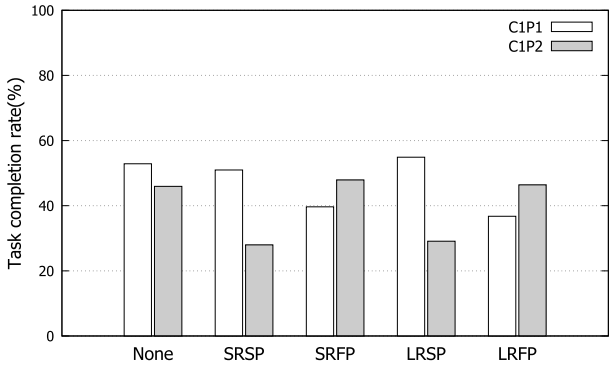


FIGURE 8. Task completion rate for C1P1 and C1P2 with various migration methods.

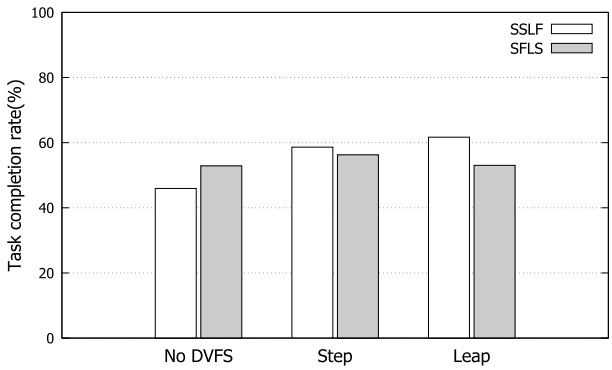


FIGURE 9. Performance of processor assignment and DVFS component on C1P1Q2.

FIGURE 10. Task stealing on C1P1 and C1P2.

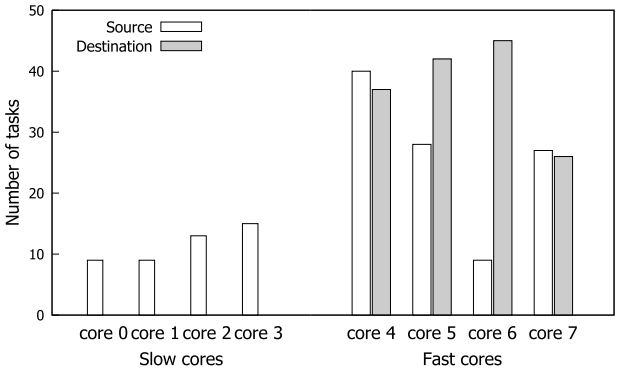


FIGURE 11. Number of task stealing on cores (C1P1S1).

respectively. In SFLS, the tasks running on a fast processor have a short execution time. The shortest task (SRFP) or longest task (LRSP) among those running on a fast processor is selected for migration to the slow cores. At the time of migration, the target core will be idle with no tasks in the queue. Usually, a short task is migrated from to a slow core and the migrated task may not hinder the execution of a new task when it arrives onto the target core. More tasks can be completed due to the higher core utilization. The LRSP migration method performed slightly better than the no migration method. SSLF obtained the opposite results when combined with SRSP and LRSP. Long tasks running on a fast processor are migrated to a slow processor, which may lead to a longer queueing time for short tasks when they arrive. The newly arrived short task has an imminent deadline, so the task will be dropped while waiting for execution. The overheads of task migration in this environment are not critical because chip migration does not occur. If the system has high migration overheads (e.g., a distant migration target in a data center), then the option may be reconsidered or redesigned.

5) DVFS

A higher clock frequency level means greater performance by the processor, the processing of more tasks, and high power consumption. Fig. 9 shows the performance of the DVFS components on C1P1Q2. The step option had a 56.1% higher task completion rate with 23.13% more energy consumption compared with the no DVFS method. The leap option consumed slightly less energy than the step option and it enhanced the task completion rate slightly more. This result demonstrates that radical changes in the clock frequency to match the current situation is more effective than gradual changes. The DVFS component affects the processor assignment component. Using DVFS, SSLF outperformed SFLS. The task completion rates with SSLF and SFLS using different DVFS methods are shown in Fig. 9. According to the results, slow cores operated at the lowest frequency and fast cores operated at the highest frequency, where slow cores had the ability to operate at the highest frequency with DVFS. Using SSLF, short tasks are assigned to slow cores and more short tasks can be completed with a higher frequency. SFLS assigns long tasks to slow cores but the task cannot be shortened drastically even if slow cores start to run at a higher frequency.

6) TASK STEALING

The task completion rates of C1P1 and C1P2 with the task stealing component are shown in Fig. 10. Idle cores with no tasks in the queue have the opportunity to steal a task and as a general rule, it is expected that the task completion rate will be enhanced by reducing the idle time and achieving better core utilization. However, the task

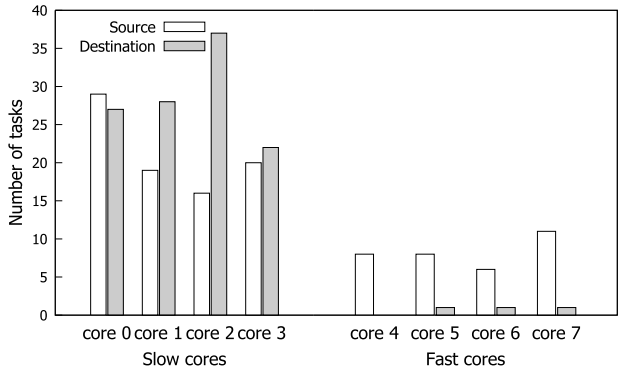


FIGURE 12. Number of task stealing on cores (C1P2S1).

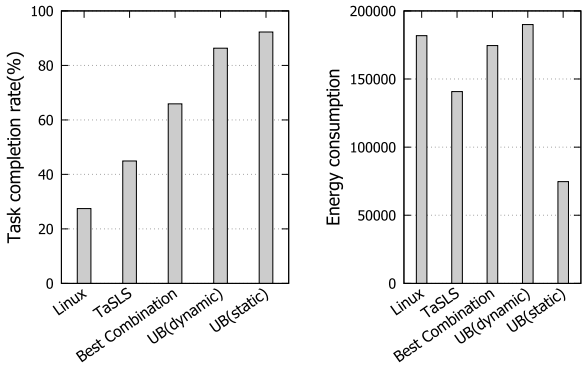


FIGURE 13. Task completion rate and energy consumption.

stealing method had a harmful effect on every combination, where task completion rate actually decreased. Fig. 11 and Fig. 12 show the source and destination for task stealing with SFLS and SSLF, respectively. Using SFLS, task stealing occurred mostly from fast cores to fast cores, although a few tasks came from slow cores with long tasks. This stealing of long tasks made the destination core busy for a long period and incoming tasks could not tolerate the queueing time. The same issue occurred with SSLF, where the only difference was the processor affected.

7) BEST COMBINATION

Overall task completion rate and energy consumption are shown in Fig. 13. The best combination (C1P2Q2M2F2) achieved the task completion rate of 65.66% and the energy consumption of 174856 energy units. The best combination method classifies the tasks by the average method, and has SSLF processor assignment, SRF queue ordering, LRFP migration method, and the Leap DVFS method. Compared with the task completion rate by the best algorithm (TaSLS) from previous study [9], the best combination had a 46.1% higher task completion rate. The energy consumption rate was also higher than the previous method, but the best combination consumed only 24.2% more energy. Energy consumption per task completion (ECTC) is shown in Fig. 13 to evaluate energy efficiency. The best combination was 13.5% better than TaSLS. According to the result, SSLF was better

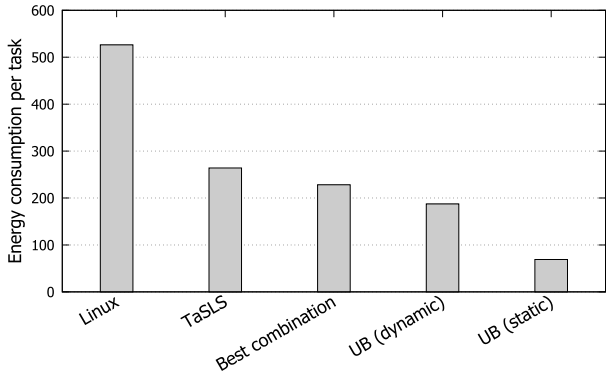


FIGURE 14. Energy consumption per task.

TABLE 5. Performance difference of TaSLS and proposed best combination using Mann-Whitney test.

Performance metric	Significance p-value	Result
Task completion rate	0.009	Accept H_1
Energy consumption per task	0.028	Accpet H_1

when DVFS method is applied, and task stealing affected performance loss. The task classification method of TaSLS is SFLS and DVFS method is connected by load status. The best combination supported the previous analysis of each component. In summary, the completion time of a short task determined the performance. Thus, a newly arrived short task should be executed as soon as possible to satisfy its deadline; otherwise, the task will be dropped. For statistical significance, the performance between TaSLS and proposed best combination were evaluated using non-parametric Mann-Whitney test to make a decision to accept or reject a hypothesis. The Mann-Whitney test is used to evaluate two different algorithms.

- Mann-Whitney test for Table 5:
  - H_0: Performance difference not exists between TaSLS and proposed best combination.
  - H_1: Performance difference exists between TaSLS and proposed best combination.

The results of Mann-Whitney test for 95% significance level ($p < 0.05$) are shown in Table 5, which confirms that we can reject H_0 (p-values are below 0.05). This means proposed algorithm outperformed in terms of both task completion rate and energy consumption per task.

8) UPPER BOUND AND ENERGY BOUND

The dynamic and proposed upper bound method are compared in Fig. 13. The static method had a higher completion rate and it made great energy savings. The dynamic method can be used without prior knowledge of each task's arrival time. It is possible to obtain a tighter upper bound for the task completion rate using the dynamic method. However, the dynamic method does not consider energy consumption. If prior knowledge of every task's arrival time is available, then the energy bound can be obtained by the

proposed method. Fig. 13 shows the energy consumption by the combined best method and the energy bound determined by the proposed method. The energy bound consumed 47.2% of the energy by the combined best algorithm, i.e., the energy consumption was improved greatly.

VI. CONCLUSION

In this study, we analyzed previous algorithms and the environment to extract important components in an attempt to construct the best task scheduling method for an example system. The components include task classification, processor assignment, queue ordering, task migration, DVFS, and task stealing. Various combinations of the components using different options are analyzed and the best combination is found. The best combination (C1P2Q2M2F2) achieved a 13.5% higher energy efficiency (energy consumption per completed task) than previous best task scheduling method. Our study of identifying important components and combining those components to construct an intelligent task scheduler can enhance the overall performance (time and energy-wise) of an example system. This research will be a stepping stone to similar research and may be applied to various systems for performance enhancement. For example, this methodology can be applied to systems with different core architectures. In addition, we proposed a energy bound estimation that determines the lower bound for energy consumption under a given task completion rate.

REFERENCES

[1] W. Zhang, E. Bai, H. He, and A. M. K. Cheng, "Solving energy-aware real-time tasks scheduling problem with shuffled frog leaping algorithm on heterogeneous platforms," *Sensors*, vol. 15, no. 6, pp. 13778–13804, 2015.

[2] R. Kumar, D. M. Tullsen, N. P. Jouppi, and P. Ranganathan, "Heterogeneous chip multiprocessors," *Computer*, vol. 38, no. 11, pp. 32–38, 2005.

[3] C.-C. Lin, Y.-C. Syu, C.-J. Chang, J.-J. Wu, P. Liu, P.-W. Cheng, and W.-T. Hsu, "Energy-efficient task scheduling for multi-core platforms with per-core DVFS," *J. Parallel Distrib. Comput.*, vol. 86, pp. 71–81, Dec. 2015.

[4] T. D. Burd and R. W. Brodersen, "Energy efficient CMOS microprocessor design," in *Proc. 28th Annu. Hawaii Int. Conf. Syst. Sci.*, vol. 1, Jan. 1995, pp. 288–297.

[5] *Enhanced Intel SpeedStep Technology for the Intel Pentium M Processor—White Paper*, Intel, Santa Clara, CA, USA, 2004.

[6] J. A. Winter, D. H. Albonesi, and C. A. Shoemaker, "Scalable thread scheduling and global power management for heterogeneous many-core architectures," in *Proc. 19th Int. Conf. Parallel Archit. Compilation Techn.*, Sep. 2010, pp. 29–39.

[7] D. T. Marr, F. Binns, D. L. Hill, G. Hinton, D. A. Koufaty, J. A. Miller, and M. Upton, "Hyper-threading technology architecture and microarchitecture," *Intel Technol. J.*, vol. 6, no. 1, pp. 1–12, 2002.

[8] J.-K. Kim, S. Shivle, H. J. Siegel, A. A. Maciejewski, T. D. Braun, M. Schneider, S. Tideman, R. Chitta, R. B. Dilmaghani, R. Joshi, A. Kaul, A. Sharma, S. Sripada, P. Vangari, and S. S. Yellampalli, "Dynamically mapping tasks with priorities and multiple deadlines in a heterogeneous environment," *J. Parallel Distrib. Comput.*, vol. 67, pp. 154–169, Feb. 2007.

[9] S. I. Kim, H. T. Kim, G. S. Kang, and J.-K. Kim, "Using DVFS and task scheduling algorithms for a hard real-time heterogeneous multicore processor environment," in *Proc. Workshop Energy Efficient High Perform. Parallel Distrib. Comput.*, 2013, pp. 23–30.

[10] Y. Yu and V. K. Prasanna, "Power-aware resource allocation for independent tasks in heterogeneous real-time systems," in *Proc. IEEE 9th Int. Conf. Parallel Distrib. Syst.*, Dec. 2002, pp. 341–348.

[11] C.-M. Hung, J.-J. Chen, and T.-W. Kuo, "Energy-efficient real-time task scheduling for a DVS system with a non-DVS processing element," in *Proc. IEEE Real-Time Syst. Symp.*, Dec. 2006, pp. 303–312.

[12] J. L. Henning, "SPEC CPU2006 benchmark descriptions," *ACM SIGARCH Comput. Archit. News*, vol. 34, no. 4, pp. 1–17, 2006.

[13] J.-J. Chen, A. Schranzhofer, and L. Thiele, "Energy minimization for periodic real-time tasks on heterogeneous processing units," in *Proc. IEEE Int. Symp. Parallel Distrib. Process.*, May 2009, pp. 1–12.

[14] L. Wenjing and W. Lisheng, "Energy-considered scheduling algorithm based on heterogeneous multi-core processor," in *Proc. Int. Conf. Mechatronic Sci., Electr. Eng. Comput. (MEC)*, Aug. 2011, pp. 1151–1154.

[15] J. M. Calandrino, D. Baumberger, T. Li, S. Hahn, and J. H. Anderson, "Soft real-time scheduling on performance asymmetric multicore platforms," in *Proc. 13th IEEE Real Time Embedded Technol. Appl. Symp.*, Apr. 2007, pp. 101–112.

[16] H.-K. Tang, P. Ramanathan, and K. Compton, "Combining hard periodic and soft aperiodic real-time task scheduling on heterogeneous compute resources," in *Proc. Int. Conf. Parallel Process.*, Sep. 2011, pp. 753–762.

[17] J. Lin, A. Srivatsa, A. Gerstlauer, and B. L. Evans, "Heterogeneous multiprocessor mapping for real-time streaming systems," in *Proc. IEEE Int. Conf. Acoust., Speech Signal Process. (ICASSP)*, May 2011, pp. 1605–1608.

[18] J.-Q. Li, S.-C. Bai, P.-Y. Duan, H.-Y. Sang, Y.-Y. Han, and Z.-X. Zheng, "An improved artificial bee colony algorithm for addressing distributed flow shop with distance coefficient in a prefabricated system," *Int. J. Prod. Res.*, Feb. 2019. doi: 10.1080/00207543.2019.1571687.

[19] Y. Han, J.-Q. Li, D. Gong, and H. Sang, "Multi-objective migrating birds optimization algorithm for stochastic lot-streaming flow shop scheduling with blocking," *IEEE Access*, vol. 7, pp. 5946–5962, 2019.

[20] Y. Han, D. Gong, Y. Jin, and Q. Pan, "Evolutionary multiobjective blocking lot-streaming flow shop scheduling with machine breakdowns," *IEEE Trans. Cybern.*, vol. 49, no. 1, pp. 184–197, Jan. 2019.

[21] Z.-X. Zheng, J.-Q. Li, and P.-Y. Duan, "Optimal chiller loading by improved artificial fish swarm algorithm for energy saving," *Math. Comput. Simul.*, vol. 155, pp. 227–243, Jan. 2019.

[22] S. K. Baruah, V. Bonifaci, R. Bruni, and A. Marchetti-Spaccamela, "ILP models for the allocation of recurrent workloads upon heterogeneous multiprocessors," *J. Scheduling*, vol. 22, no. 2, pp. 195–209, 2019.

[23] S. Moulik, R. Devaraj, and A. Sarkar, "HEART: A heterogeneous energy-aware real-time scheduler," in *Proc. 32nd Int. Conf. VLSI Design 18th Int. Conf. Embedded Syst. (VLSID)*, Jan. 2019, pp. 476–481.

[24] S. Moulik, R. Devaraj, and A. Sarkar, "HEALERS: A heterogeneous energy-aware low-overhead real-time scheduler," *IET Comput. Digit. Techn.*, Jun. 2019. doi: 10.1049/iet-cdt.2019.0023.

[25] A. Mazouz, A. Laurent, B. Pradelle, and W. Jalby, "Evaluation of CPU frequency transition latency," *Comput. Sci.-Res. Develop.*, vol. 29, nos. 3–4, pp. 187–195, 2014.

[26] H. S. Chwa, J. Seo, J. Lee, and I. Shin, "Optimal real-time scheduling on two-type heterogeneous multicore platforms," in *Proc. IEEE Real-Time Syst. Symp.*, Dec. 2015, pp. 119–129.

[27] S. Moulik, R. Devaraj, and A. Sarkar, "HETERO-SCHED: A low-overhead heterogeneous multi-core scheduler for real-time periodic tasks," in *Proc. IEEE 20th Int. Conf. High Perform. Comput. Commun., IEEE 16th Int. Conf. Smart City, IEEE 4th Int. Conf. Data Sci. Syst. (HPCC/SmartCity/DSS)*, Jun. 2018, pp. 659–666.

[28] S. Moulik, R. Devaraj, and A. Sarkar, "COST: A cluster-oriented scheduling technique for heterogeneous multi-cores," in *Proc. IEEE Int. Conf. Syst., Man, Cybern. (SMC)*, Oct. 2018, pp. 1951–1957.

[29] M. A. Awan, P. M. Yomsi, G. Nelissen, and S. M. Petters, "Energy-aware task mapping onto heterogeneous platforms using DVFS and sleep states," *Real-Time Syst.*, vol. 52, no. 4, pp. 450–485, 2016.

[30] R. Ayari, I. Hafnaoui, G. Beltrame, and G. Nicolescu, "ImGA: An improved genetic algorithm for partitioned scheduling on heterogeneous multi-core systems," *Des. Automat. Embedded Syst.*, vol. 22, nos. 1–2, pp. 183–197, 2018.

[31] R.-M. Chen, Y.-M. Shen, and C.-T. Wang, "Ant colony optimization inspired swarm optimization for grid task scheduling," in *Proc. Int. Symp. Comput., Consum. Control (IS3C)*, Jul. 2016, pp. 461–464.

[32] A. Abdi and H. R. Zarandi, "A meta heuristic-based task scheduling and mapping method to optimize main design challenges of heterogeneous multiprocessor embedded systems," in *Microelectron. J.*, vol. 87, pp. 1–11, May 2019.
[33] T. Zhang, X. Li, and G. Liu, "An improved artificial bee colony algorithm for the task assignment in heterogeneous multicore architectures," in *Proc. Int. Conf. Swarm Intell.* Cham, Switzerland: Springer, 2018, pp. 179–187.
[34] E. G. Coffman, *Computer and Job-Shop Scheduling Theory*. New York, NY, USA: Wiley, 1976.
[35] D. Fernandez-Baca, "Allocating modules to processors in a distributed system," *IEEE Trans. Softw. Eng.*, vol. 15, no. 11, pp. 1427–1436, Nov. 1989.
[36] O. H. Ibarra and C. E. Kim, "Heuristic algorithms for scheduling independent tasks on nonidentical processors," *J. ACM*, vol. 24, no. 2, pp. 280–289, Apr. 1977.

SUNG IL KIM received the B.S. degree in electrical engineering from Korea University, in 2011, where he is currently pursuing the Ph.D. degree in computer engineering. His research interests include heterogeneous distributed computing, real-time scheduling, evolutionary algorithms, and machine learning.

JONG-KOOK KIM received the B.S. degree in electronic engineering from Korea University, Seoul, South Korea, in 1998, and the M.S. and Ph.D. degrees from the School of Electrical and Computer Engineering, Purdue University, in 2000 and 2004, respectively. He is currently a Professor with the School of Electrical Engineering, Korea University, where he joined, in 2007. He was with the Samsung SDS's IT Research and Development Center, from 2005 to 2007. His research interests include heterogeneous distributed computing, energy-aware computing, resource management, evolutionary heuristics, distributed mobile computing, artificial neural networks, deep learning and distributed robot systems. He is a Senior Member of the ACM.